Data Visualization in R

Base graphics

R.W. Oldford

Data visualization in R

There exist several graphical systems in `R` that can be used to construct displays of nearly arbitrary complexity that can be tailored to any particular application of interest.

In this course, we will make use of only a small handful of these, primarily

- base `graphics` for quick construction and layout of standard plots,
- `grid` graphics for quick construction of layout of arbitrary plots,
- `ggplot2` for quickly specifying many useful plots in a data analysis, and finally
- `loon` to construct highly interactive and extendible graphics for exploratory data analysis(particularly for high dimensional data).

We will also make use of `shiny` for interactive presentation graphics that allow some constrained exploratory data analysis by the viewer.

There are also more than 200 other `R` packages on CRAN (including the open GL package `RGL`) that have "visual" in their description and so provide some visualization capapbilities.

Data visualization in R - some general graphics packages of interest

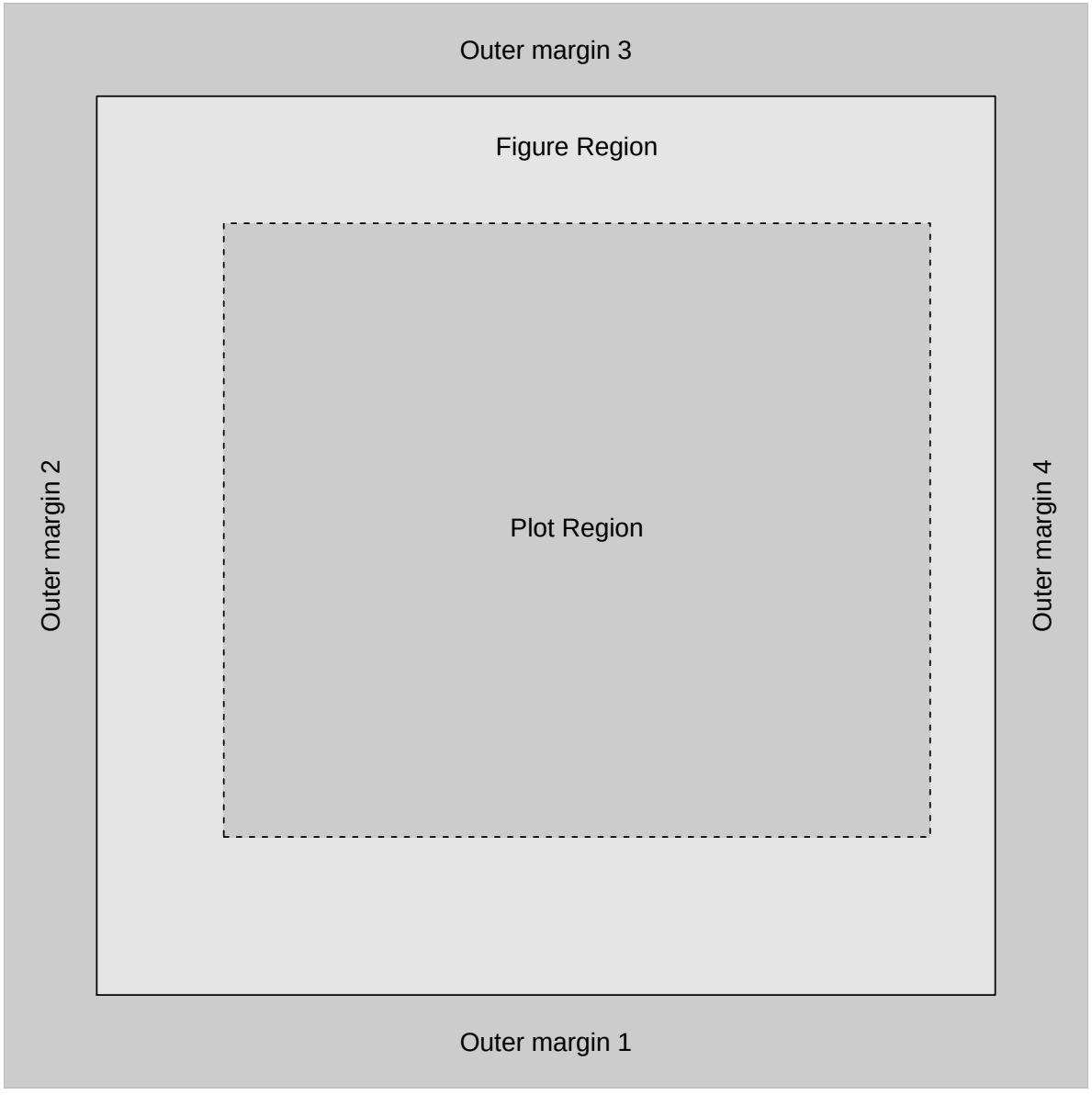
package	comments	special strengths
`graphics`	R's base graphics	simple, control of layout, well integrated into R, good for prototyping new graphics
`grid`, `gridBase`, `gridExtra`, `gtable`	R Core package, can be integrated with base graphics	classic computer graphics abstractions (viewports, coordinate systems, clipping, etc.), flexible and open-ended, excellent for prototyping (especially complex designs), arbitrary layout
`RGL`	R Core package, interface to Open GL library	classic 3D graphics based on `Open GL`(viewpoints, shading, light sources, clipping, etc.)
`ggplot2`	Implemented via `grid`, inspired by "Grammar of Graphics" model, pipeline models for graphics	part of the `tidyverse`, good for construction of presentation quality graphics, displays are easily modified as data analysis unfolds, can be used in conjunction with `gridGraphics` code
`loon`	R package for interactive data analysis, basic design implemented in `tcktk`	interactive, integrated into R, extendible, can capture and respond to nearly any mouse and/or keyboard event, arbitrary interaction and layout via `tcltk` functionality
`shiny`	Web browser based *r*eactive graphics	arbitrary layout, filters, and displays.

base graphics

This is the original graphics system design, dating back to the original S language, and consequently those most embedded in R and its various statistical and analysis methods.

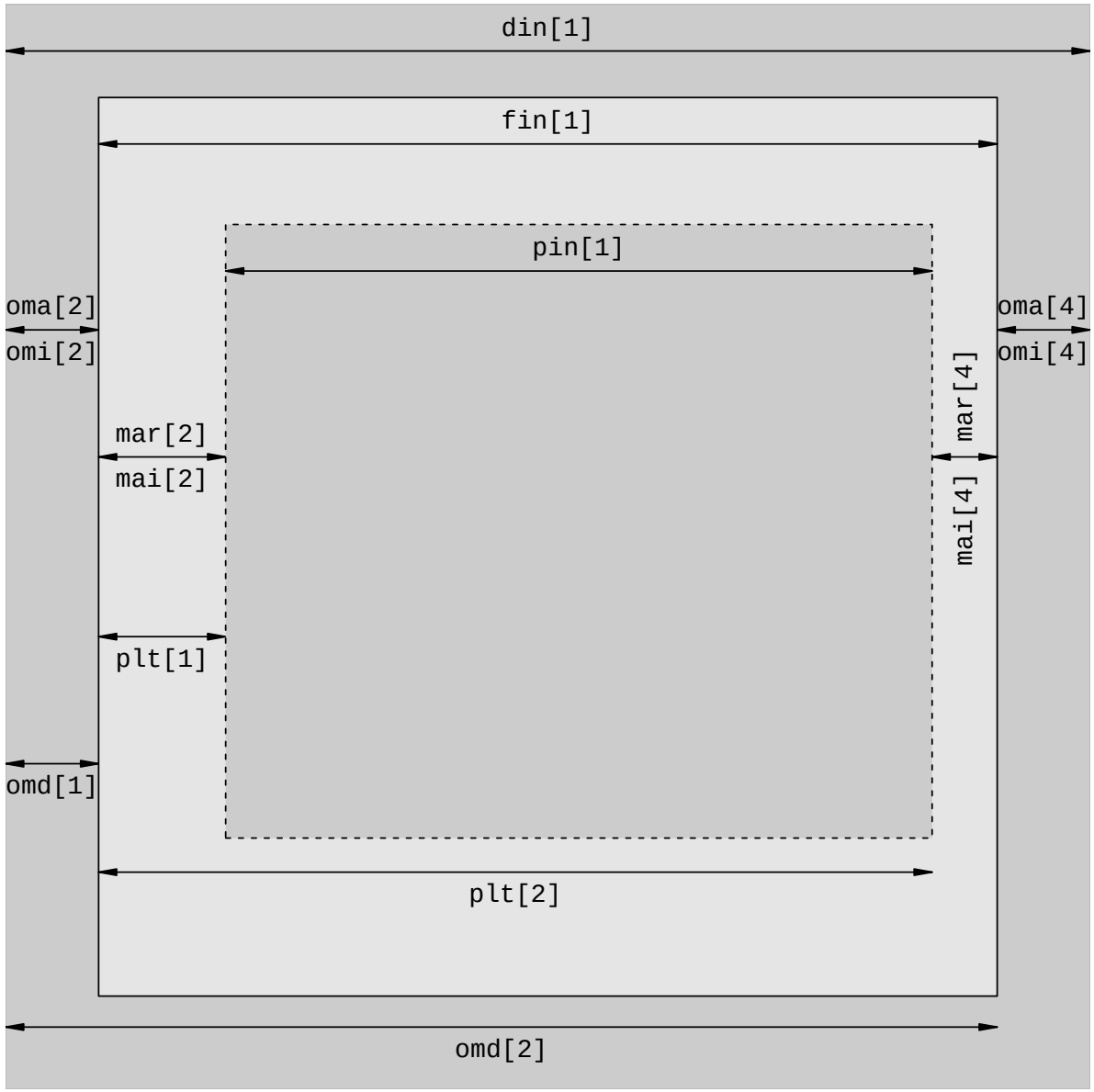
Statistical plotting functions	`plot`, `barplot`, `boxplot`, `assocplot`, `cdplot`, `contour`, `filled.contour`, `coplot`, `dotchart`, `fourfoldplot`, `hist`, `matlines`, `matplot`, `matpoints`, `mosaicplot`, `pairs`, `pie`, `rug`, `smoothScatter`, `spineplot`, `stars`, `stem`, `stripchart`, `sunflowerplot`, `symbols`
Geometric plotting	`abline`, `arrows`, `curve`, `image`, `lines`, `persp`, `points`, `polygon`, `polypath`, `rasterImage`, `rect`, `segments`, `text`
Plot arguments	`type`, `xlim`, `ylim`, `log`, `main`, `sub`, `xlab`, `ylab`, `ann`, `axes`, `frame.plot`, `asp`, `col`, `pch`, `cex`, `lwd`, `lty`
Individual plot component functions	`axis`, `axis.POSIXct`, `clip`, `axTicks`, `box`, `grid`, `legend`, `title`
Graphical parameters	`mai`, `mar`, `mex`, `mfcol`, `mfrow`, `mfg`, `oma`, `omd`, `omi`

Plotting regions for a single plot (from Paul Murrell's R Graphics (1st edition)):

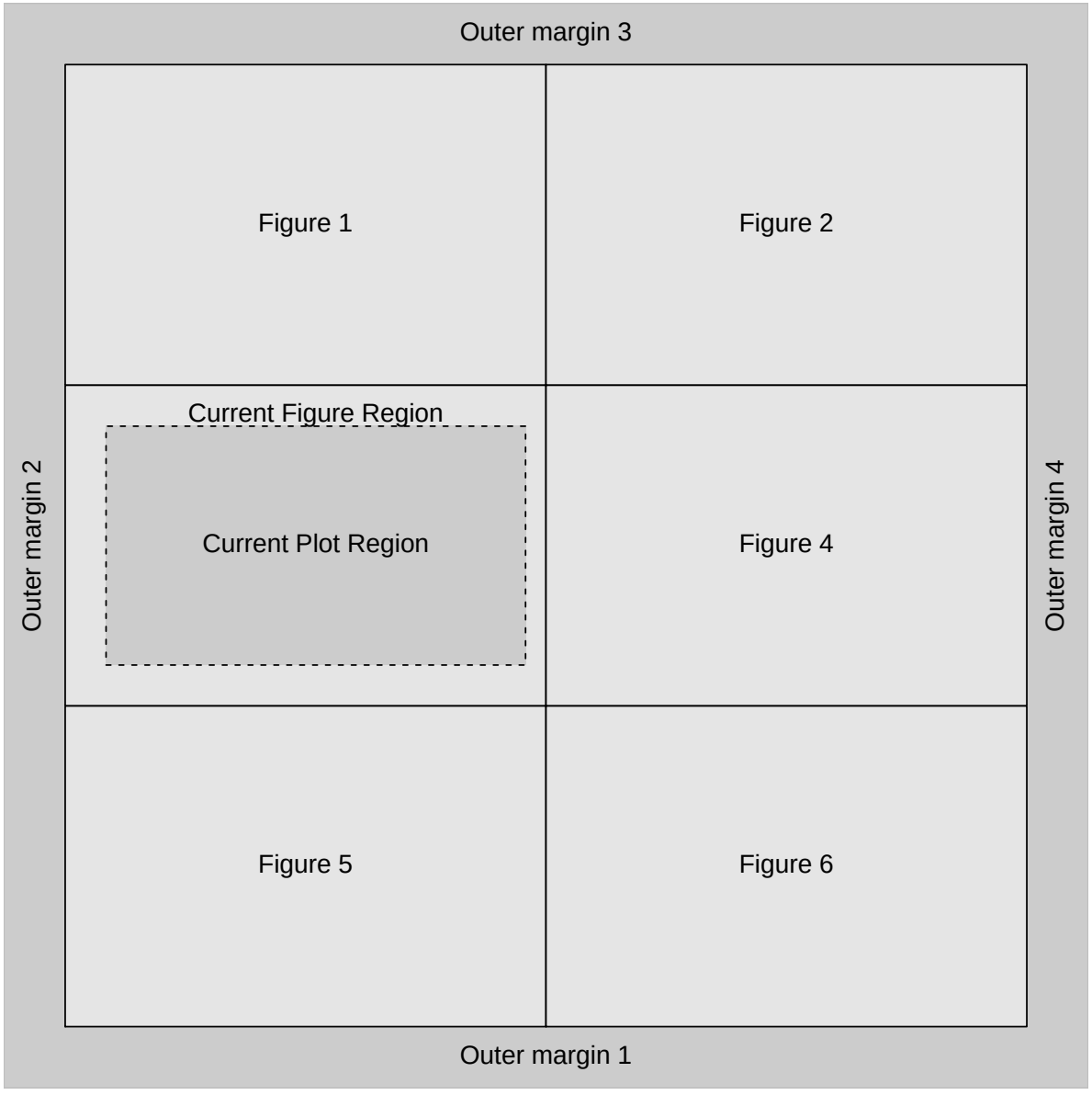


base graphics

Graphical parameters determining plotting regions (adapted from Paul Murrell's R Graphics (1st edition)):



Plotting regions for a multiple plots (from Paul Murrell's R Graphics (1st edition)):



Familiar examples: Plotting a density

```
# A density estimate
den <- density(cars$speed, bw = "SJ")
str(den)
```

```
## List of 7
##  $ x         : num [1:512] -4.97 -4.9 -4.82 -4.74 -4.67 ...
##  $ y         : num [1:512] 6.20e-05 6.70e-05 7.23e-05 7.79e-05 8.41e-05 ...
##  $ bw        : num 2.99
##  $ n         : int 50
##  $ call      : language density.default(x = cars$speed, bw = "SJ")
##  $ data.name: chr "cars$speed"
##  $ has.na    : logi FALSE
##  - attr(*, "class")= chr "density"
```

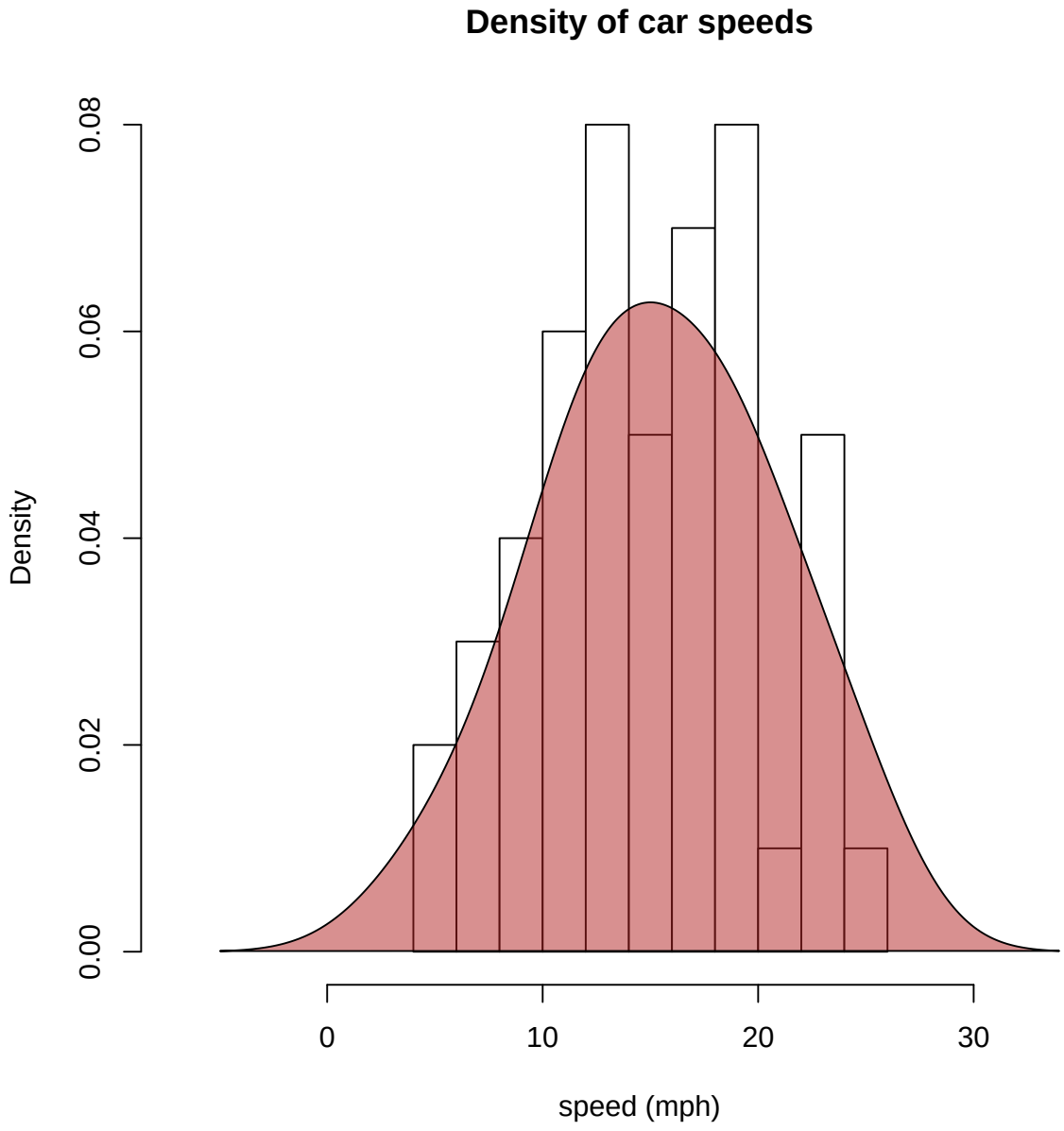
```
is.list(den)
```

```
## [1] TRUE
```

```
# The density plotted on top of a histogram
hist(cars$speed, freq = FALSE, breaks = 10,
     xlim = extendrange(den$x), col = "white",
     main ="Density of car speeds", xlab="speed (mph)")
polygon(den, col = adjustcolor("firebrick", 0.5))
```

N.B. A handy function is `xy.coords()` which tries to return plotting argument values (e.g. `x`, `y`, etc.). It is called on data given to `plot()`.

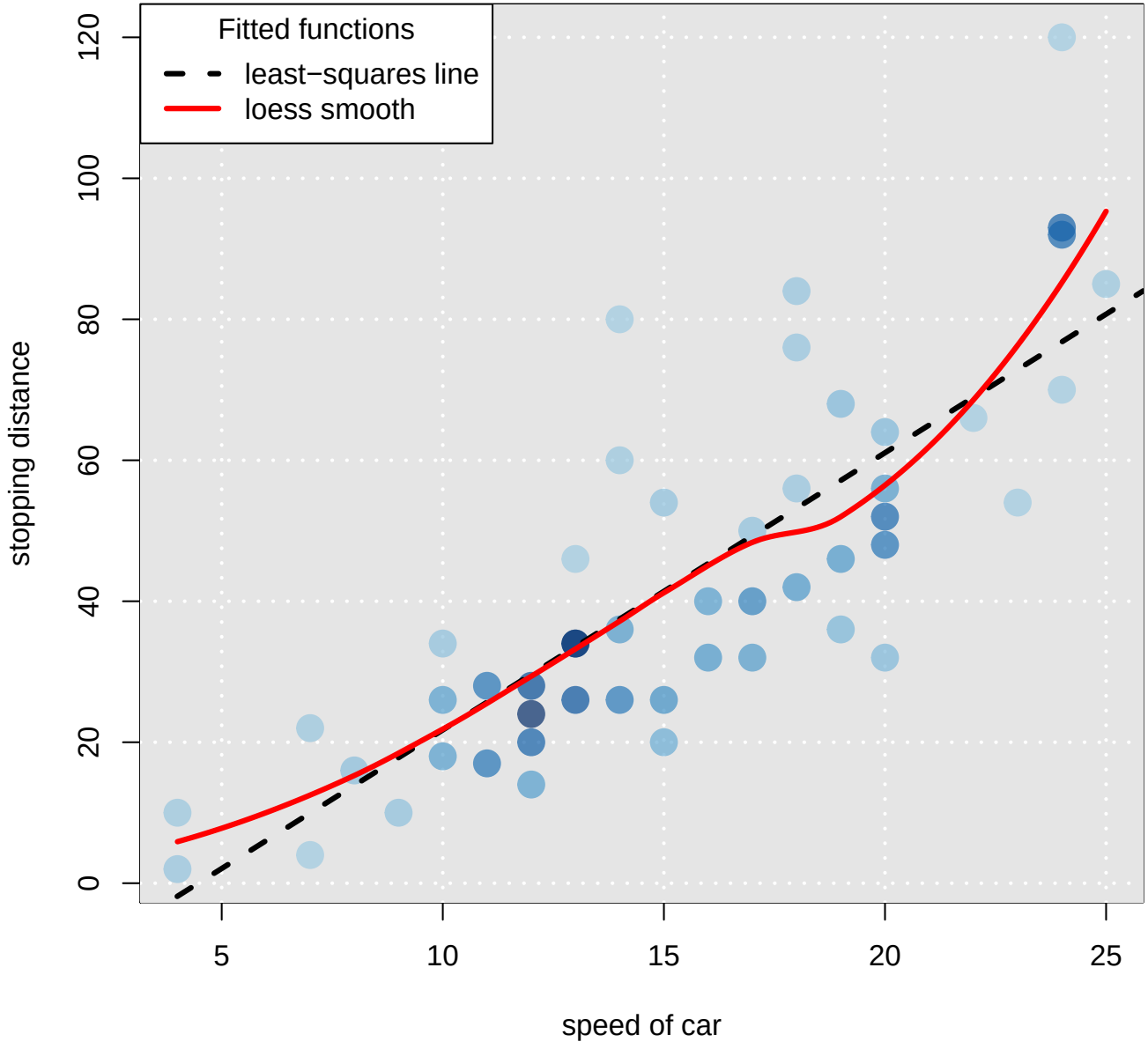
Familiar examples: Plotting a density



Familiar examples: A scatterplot

```
plot(cars$speed, cars$dist, type="n",
     xlab = "speed of car",  ylab="stopping distance")
lims <- par("usr")
xlim <- lims[1:2]
ylim <- lims[3:4]
rect(xlim[1], ylim[1], xlim[2], ylim[2], col = "grey90", border =NA)
grid(col="white", lwd = 2)
points(cars$speed, cars$dist, pch=19, cex = 2,
       col = adjustcolor(densCols(cars$speed, cars$dist), 0.7))
fit <- lm(dist ~ speed, data = cars)
abline(fit$coefficients, col = "black", lty =2, lwd=3)
sm <- loess(dist ~ speed, data = cars)
xvals <- seq(min(cars$speed), max(cars$speed), length.out=200)
lines(xvals, predict(sm, newdata = data.frame(speed = xvals)),
      col = "red", lwd =3, lty = 1)
legend("topleft", bg = "white", title = "Fitted functions",
       legend = c("least-squares line", "loess smooth"),
       col = c("black", "red"), lty = c(2, 1), lwd = c(3,3))
```

Familiar examples: A scatterplot



Familiar examples: Locations of cities in Canada

```
# A map
library(maps)
data("worldMapEnv")
str(canada.cities)
```

```
## 'data.frame':    916 obs. of  6 variables:
##  $ name       : chr  "Abbotsford BC" "Acton ON" "Acton Vale QC" "Airdrie AB" ...
##  $ country.etc: chr  "BC" "ON" "QC" "AB" ...
##  $ pop        : int  157795 8308 5153 25863 643 1090 1154 11972 1427 3604 ...
##  $ lat        : num  49.1 43.6 45.6 51.3 68.2 ...
##  $ long       : num  -122.3 -80 -72.6 -114 -135 ...
##  $ capital    : int  0 0 0 0 0 0 0 0 0 0 ...
```

```
summary(as.factor(canada.cities$capital))
```

```
##   0   1   2
## 902   1  13
```

Familiar examples: Can get the coordinates of the boundaries of Canada

```
# A map
library(maps)
data("worldMapEnv")
canada <- map("world", "Canada", plot=FALSE)
class(canada)
```

```
## [1] "map"
```

```
str(canada)
```

```
## List of 4
##  $ x    : num [1:11723] -59.8 -59.9 -60 -60.1 -60.1 ...
##  $ y    : num [1:11723] 43.9 43.9 43.9 43.9 44 ...
##  $ range: num [1:4] -141 -52.7 41.7 83.1
##  $ names: chr [1:141] "Canada:Sable Island" "Canada:5" "Canada:Grand Manan Island" "Cana
##  - attr(*, "class")= chr "map"
```

```
canada$x[1:14]
```

```
##  [1] -59.78760 -59.92227 -60.03775 -60.11426 -60.11748 -59.93604 -59.86636
##  [8] -59.72715 -59.78760        NA -66.27377 -66.32412 -66.31191 -66.25049
```

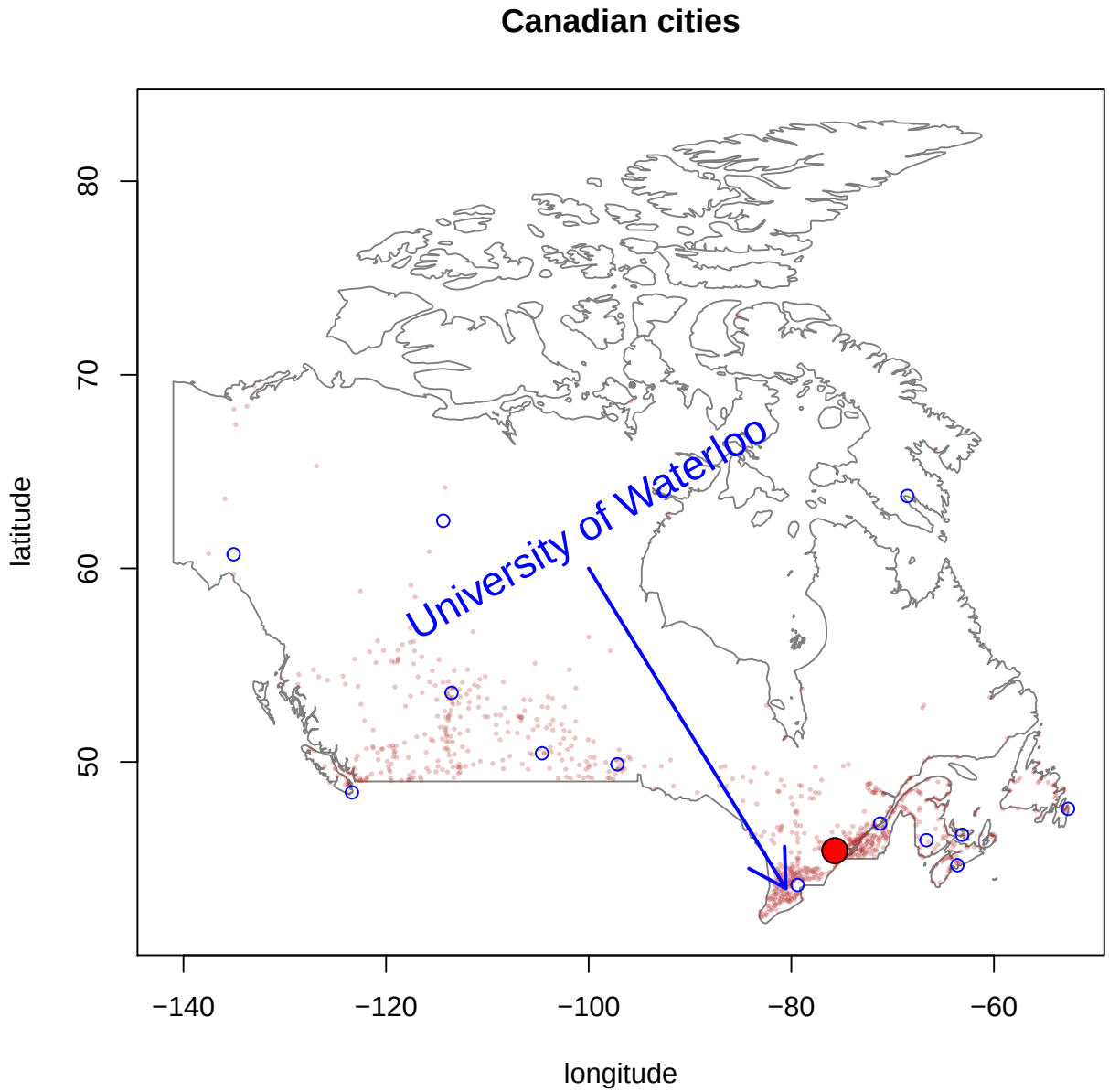
```
canada$y[1:14]
```

```
##  [1] 43.93960 43.90391 43.90664 43.93911 43.95337 43.93960 43.94717
##  [8] 44.00283 43.93960       NA 44.29229 44.25732 44.29160 44.37901
```

Familiar examples: Put the locations of the cities on the map

```
# Plot the map
plot(canada, type="l", xlab = "longitude", ylab = "latitude",
     col = "grey50", main = "Canadian cities")
not_capitals <- canada.cities$capital == 0
Ottawa <- canada.cities$capital == 1
provTerritoryCapitals <- canada.cities$capital == 2
points(canada.cities$long[not_capitals],
       canada.cities$lat[not_capitals], pch=19, cex = 0.25,
       col = adjustcolor("firebrick", 0.25))
points(canada.cities$long[provTerritoryCapitals],
       canada.cities$lat[provTerritoryCapitals], pch=21, cex = 1,
       col = "blue")
points(canada.cities$long[Ottawa],
       canada.cities$lat[Ottawa], pch=19, cex = 2,
       col = "red")
points(canada.cities$long[Ottawa],
       canada.cities$lat[Ottawa], pch=21, cex = 2,
       col = "black")
arrows(-100, 60, -80.5449, 43.4723, col="blue", lwd = 2)
text(-100, 62, "University of Waterloo", col="blue", srt = 30, cex=1.5)
```

Familiar examples: Perhaps a map of the locations of cities in Canada

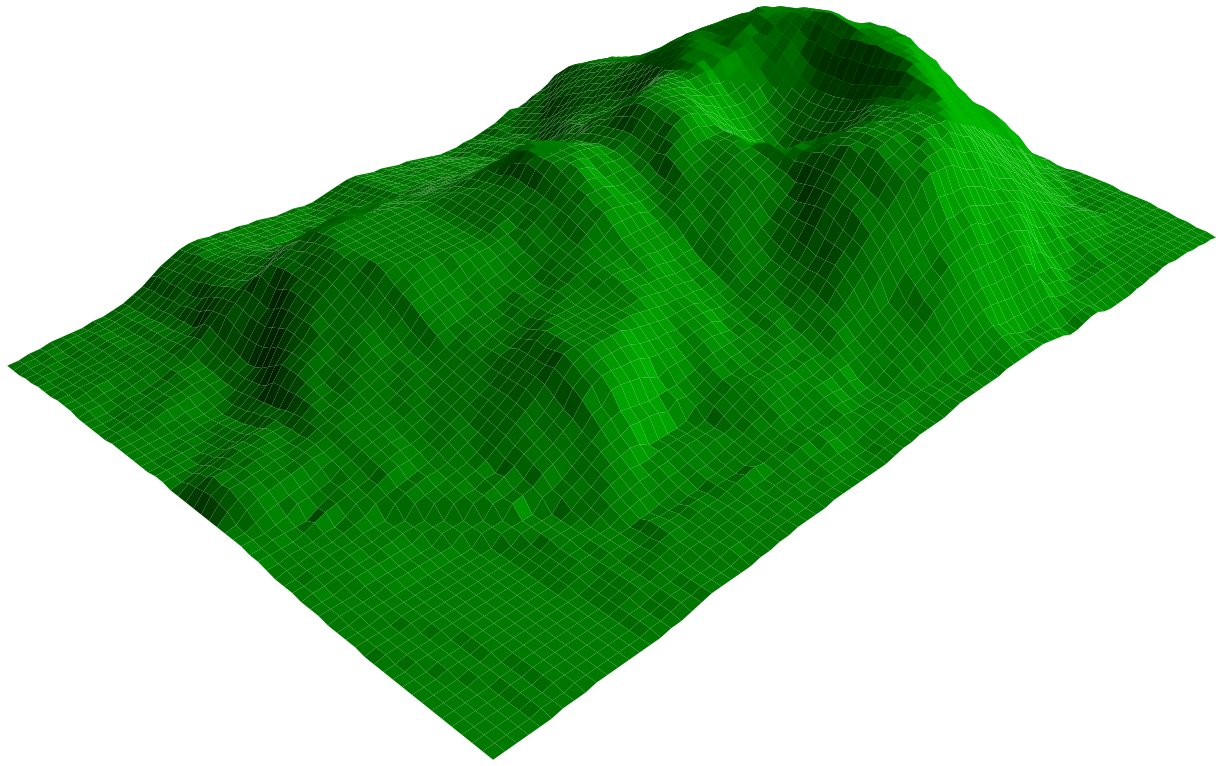


Familiar examples: A three dimensional surface, e.g. a volcano.

```
# A 3d plot of the Maunga Whau Volcano in New Zealand
z <- 2 * volcano          # Exaggerate the relief
x <- 10 * (1:nrow(z))     # 10 meter spacing (S to N)
y <- 10 * (1:ncol(z))     # 10 meter spacing (E to W)
# Don't draw the grid lines :  border = NA
persp(x, y, z, theta = 135, phi = 30, col = "green3",
      main = "Maunga Whau (Mt Eden) Volcano",
      scale = FALSE, ltheta = -120, shade = 0.75,
      border = NA, box = FALSE)
```

Familiar examples: A three dimensional surface, e.g. a volcano.

Maunga Whau (Mt Eden) Volcano



Familiar examples: Put them all together in a single display by setting the graphical parameters

Set up the graphical parameters you want (and save the old ones)

```
# Layout parameters (assignment saves previous values)
savePar <- par(mfrow=c(2, 2), cex=0.6,
               mar=c(6, 6, 2, 2), mex=0.8,
               bg="floralwhite")
```

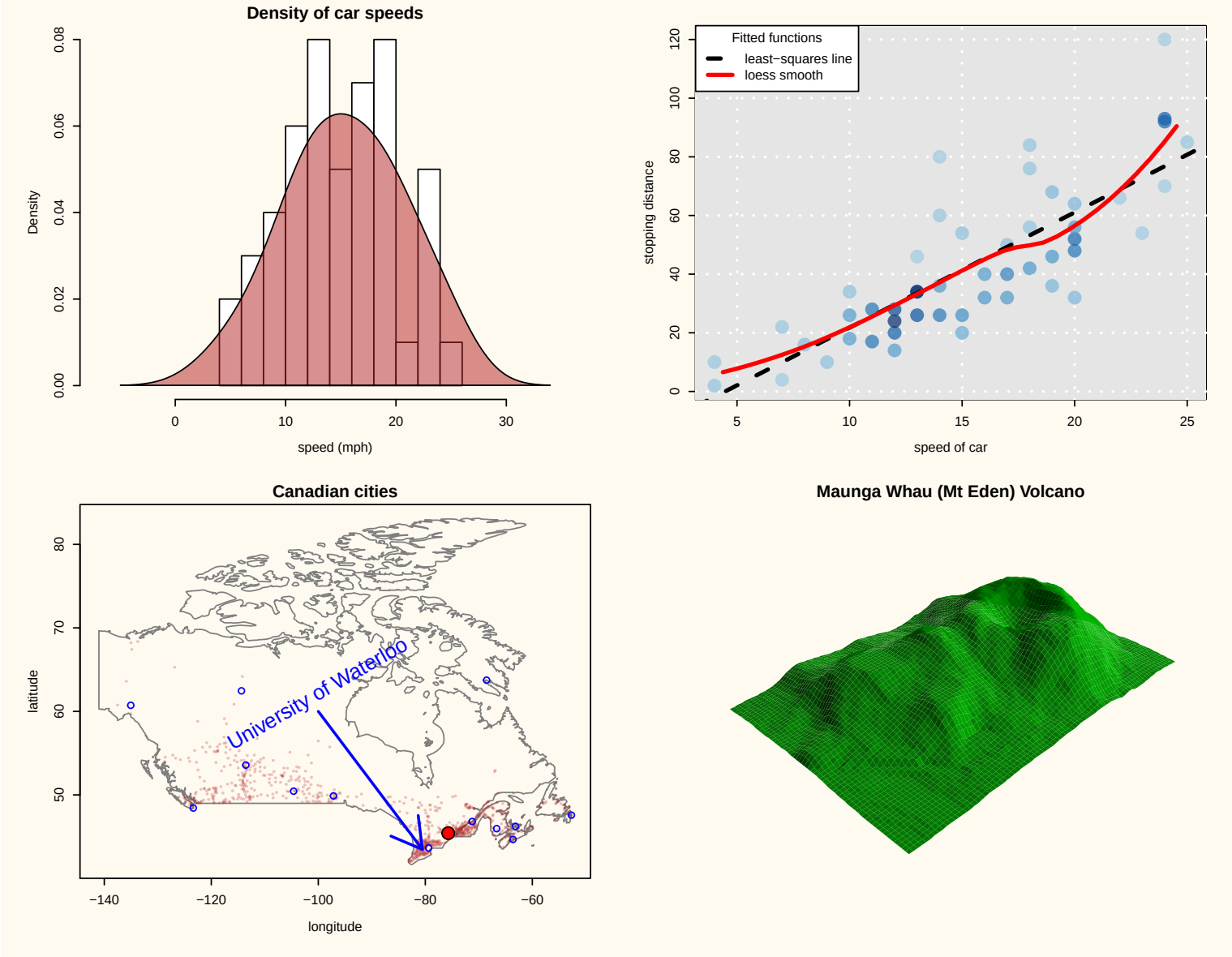
The `mfrow = c(2,2)` suggests we want to draw four plots.

Now plot each of the four plots as above, then set the graphical parameters back to their original values.

```
par(savePar)
```

base graphics

Familiar examples:



Note that the background colour is not white.

Some very powerful functions for plotting data: e.g. conditioning plots `coplot()`

```
# Tonga Trench Earthquakes
str(quakes)
```

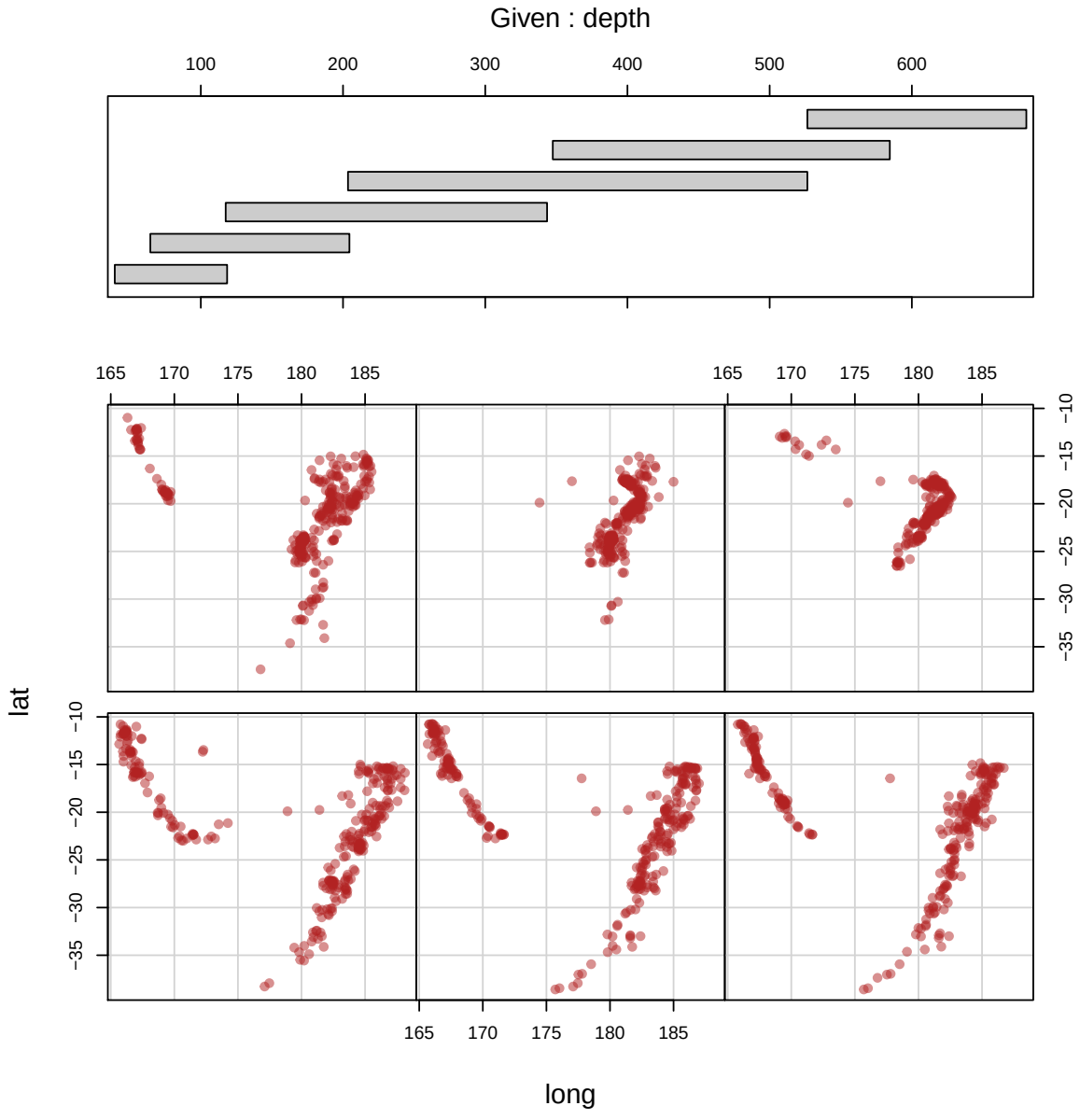
```
## 'data.frame':    1000 obs. of  5 variables:
##  $ lat     : num  -20.4 -20.6 -26 -18 -20.4 ...
##  $ long    : num  182 181 184 182 182 ...
##  $ depth   : int  562 650 42 626 649 195 82 194 211 622 ...
##  $ mag     : num  4.8 4.2 5.4 4.1 4 4 4.8 4.4 4.7 4.3 ...
##  $ stations: int  41 15 43 19 11 12 43 15 35 19 ...
```

Use a formula

```
coplot(lat ~ long | depth, data = quakes,
       pch = 19, col = adjustcolor("firebrick", 0.5))
```

base graphics

Some very powerful functions for plotting data: e.g. conditioning plots `coplot()`



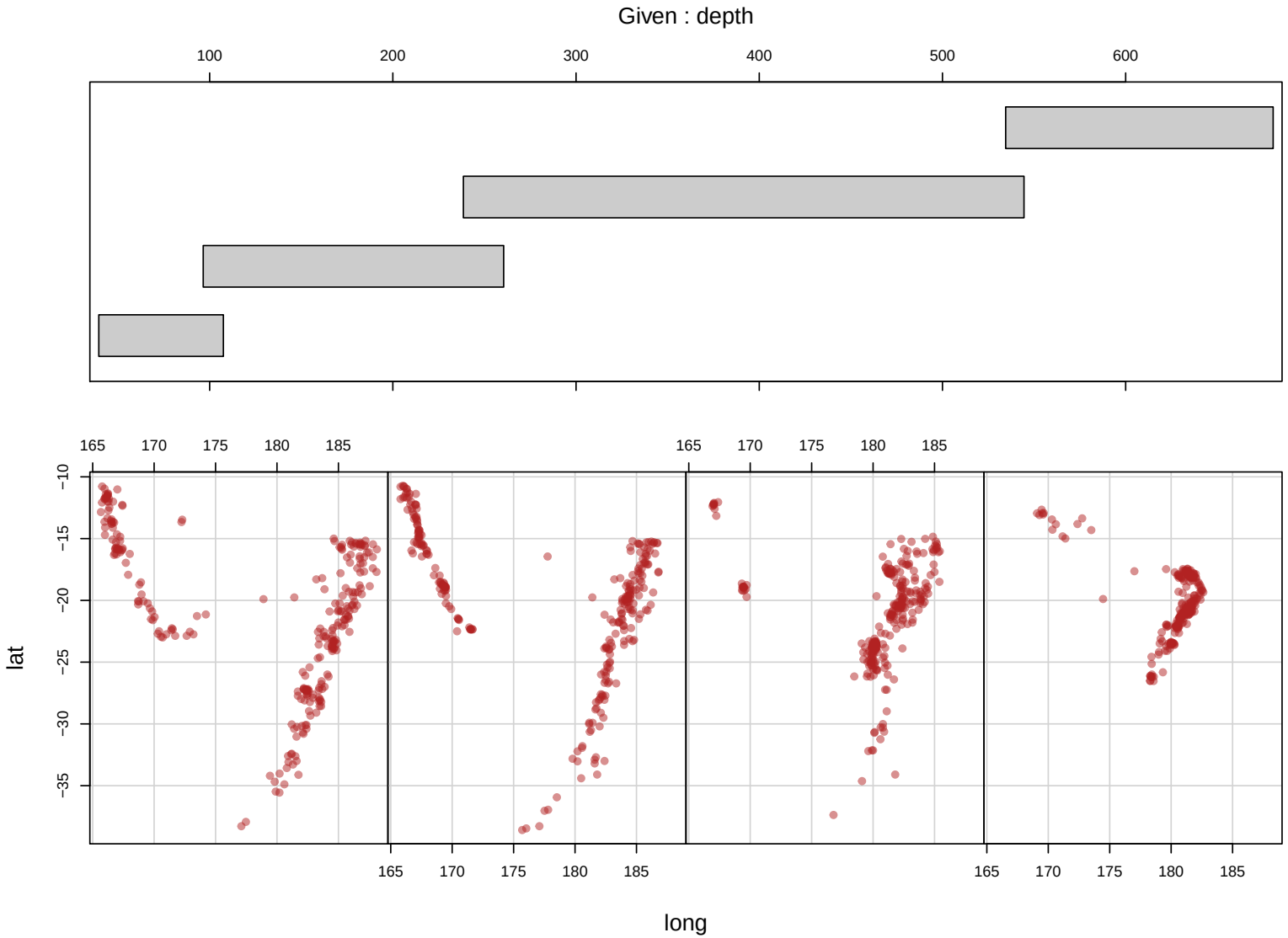
Some very powerful functions for plotting data: e.g. conditioning plots `coplot()`

- can construct your own levels to condition on (here only 4)

```
given.depth <- co.intervals(quakes$depth, number = 4, overlap = .1)
coplot(lat ~ long | depth, data = quakes,
       given.v = given.depth, rows = 1,
       pch = 19, col = adjustcolor("firebrick", 0.5))
```

Some very powerful functions for plotting data: e.g. conditioning plots `coplot()`

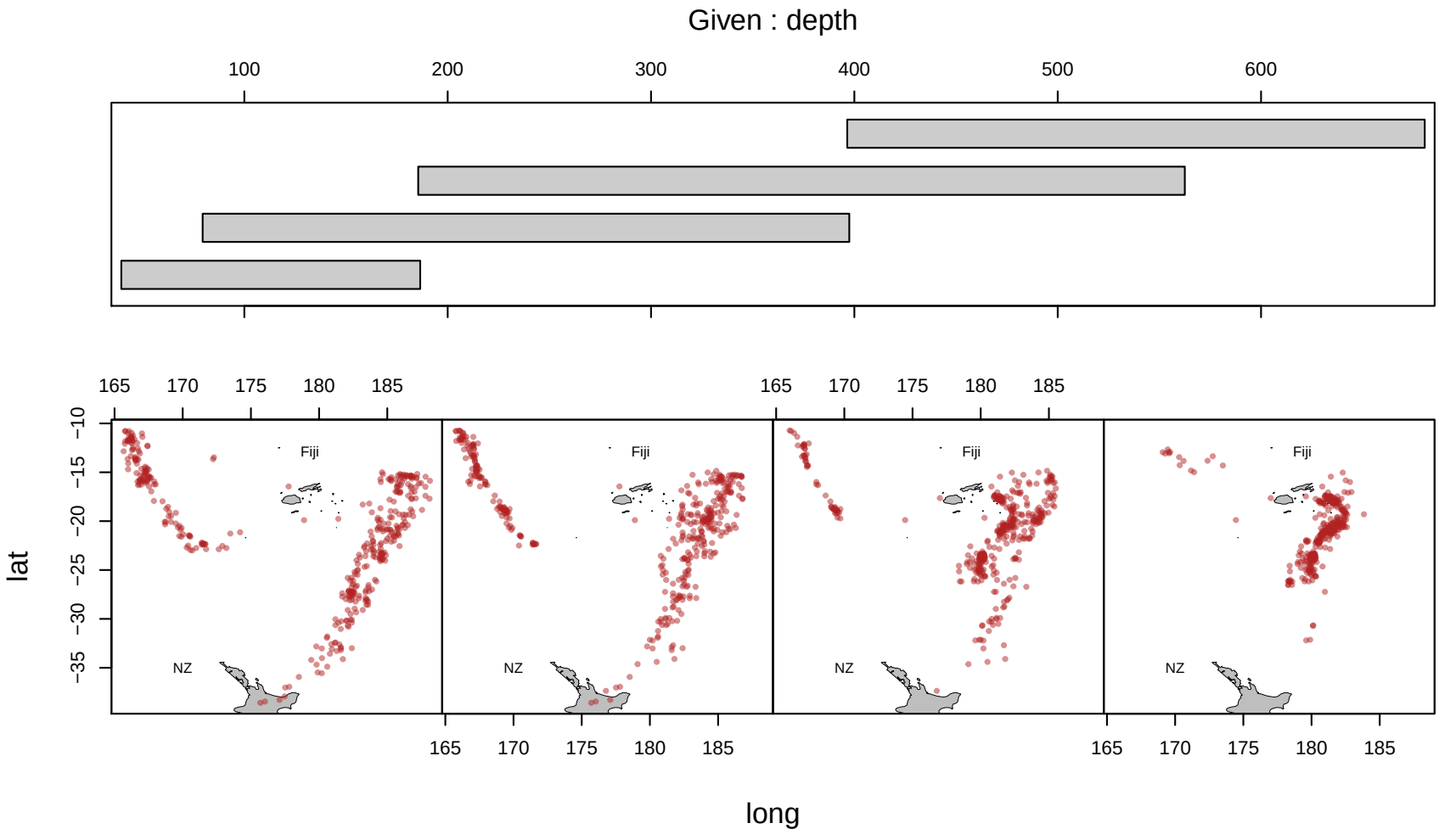
- can construct your own levels to condition on (here only 4)



A more complex example of the conditioning plots `coplot()`

```
library(maps)
coplot(lat ~ long | depth, data = quakes, number=4, rows = 1,
       panel=function(x, y, ...) {
         usr <- par("usr")
         rect(usr[1], usr[3], usr[2], usr[4], col="white")
         map("world2", regions=c("New Zealand", "Fiji"),
             add=TRUE, lwd=0.1, fill=TRUE, col="grey")
         text(180, -13, "Fiji", adj=1, cex=0.7)
         text(170, -35, "NZ", cex=0.7)
         points(x, y, pch = 19, cex = 0.5,
                col = adjustcolor("firebrick", 0.5))
       })
```

A more complex example of the conditioning plots `coplot()`



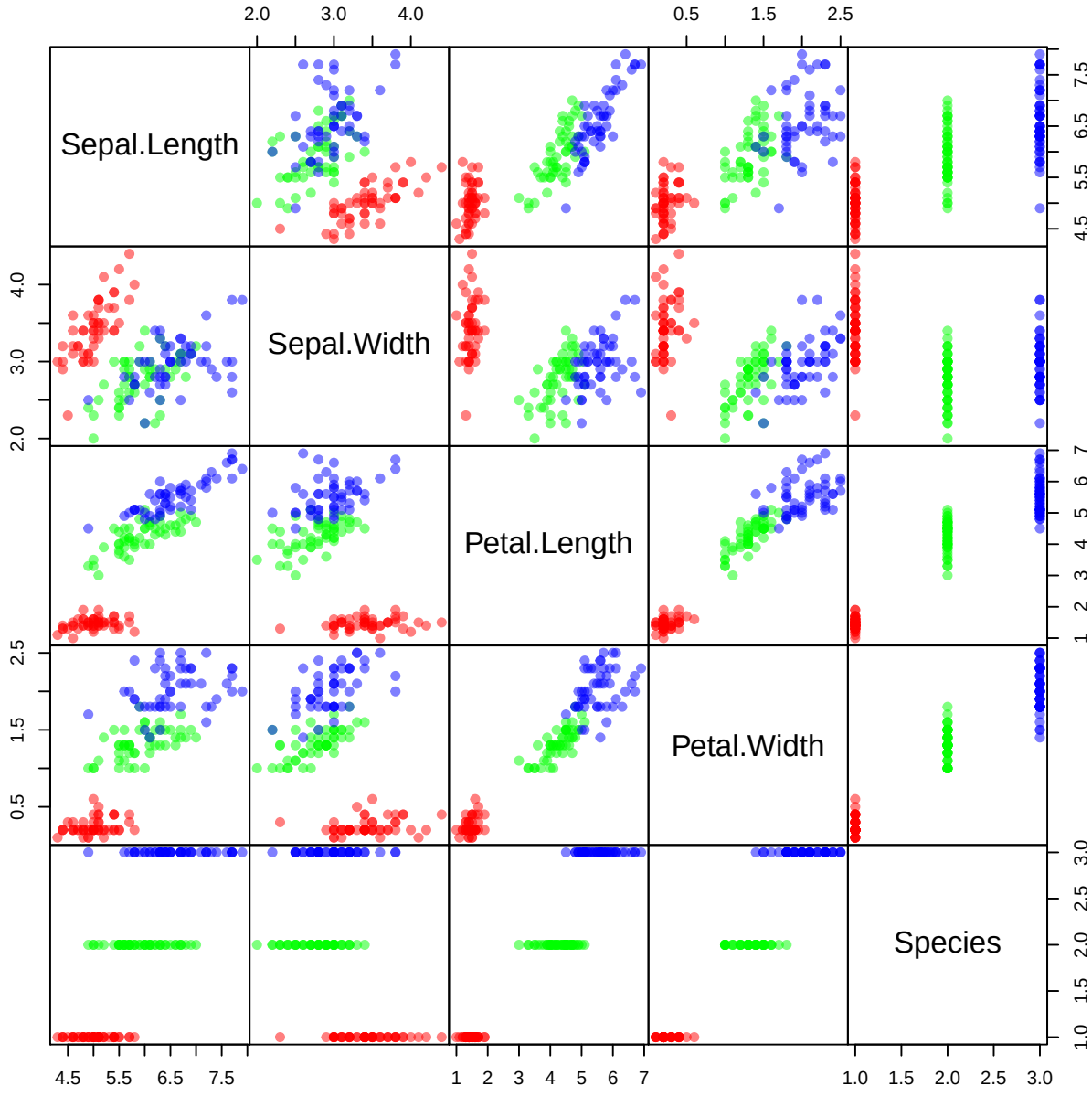
Specialized `plot()` functionality: plot multivariate data

```
str(iris)
```

```
## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1
```

```
plot(iris, gap = 0, pch = 19,
     col = adjustcolor(rainbow(3), 0.5)[iris$Species])
```

Specialized `plot()` functionality: plot multivariate data



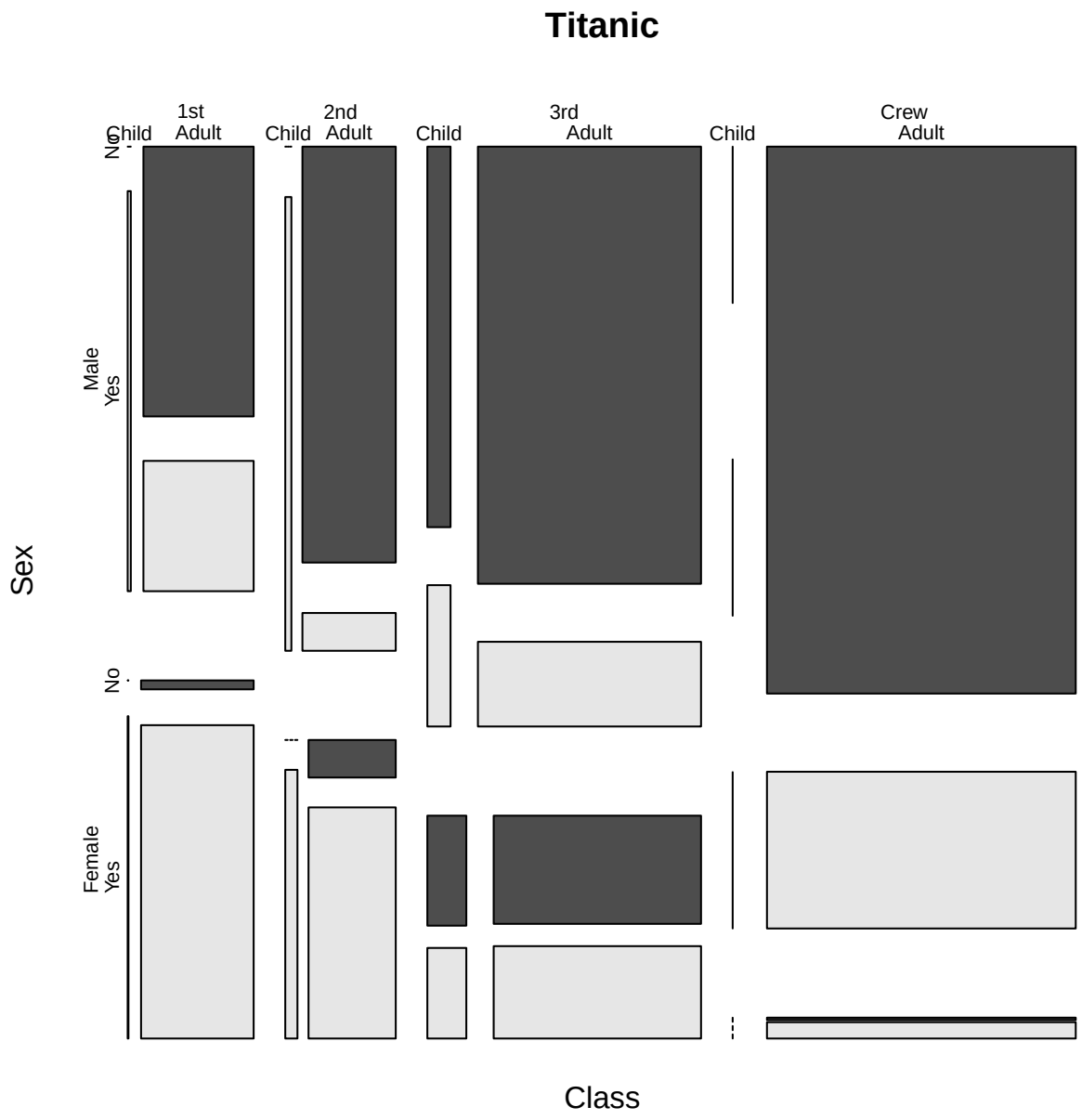
Specialized `plot()` functionality: plot categorical data

```
str(Titanic)
```

```
##  'table' num [1:4, 1:2, 1:2, 1:2] 0 0 35 0 0 0 17 0 118 154 ...
##  - attr(*, "dimnames")=List of 4
##   ..$ Class   : chr [1:4] "1st" "2nd" "3rd" "Crew"
##   ..$ Sex     : chr [1:2] "Male" "Female"
##   ..$ Age     : chr [1:2] "Child" "Adult"
##   ..$ Survived: chr [1:2] "No" "Yes"
```

```
plot(Titanic, color = TRUE)
```

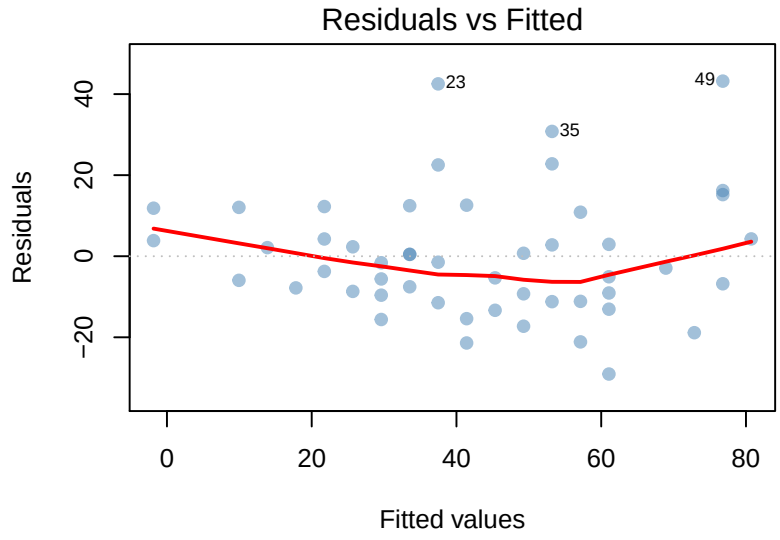
Specialized `plot()` functionality: plot categorical data

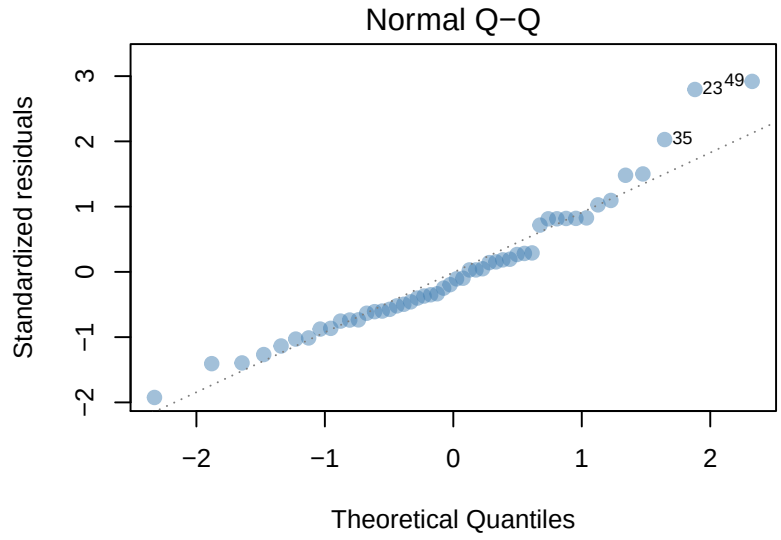


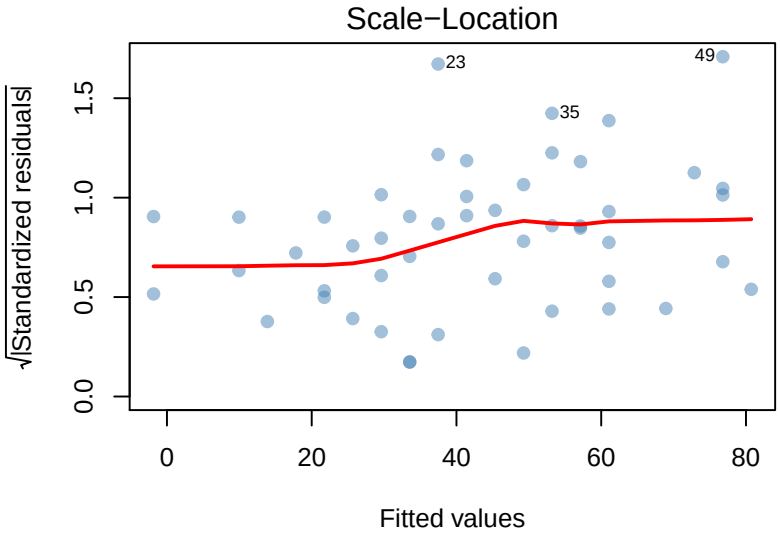
Specialized `plot()` functionality: plotting a least-squares fit

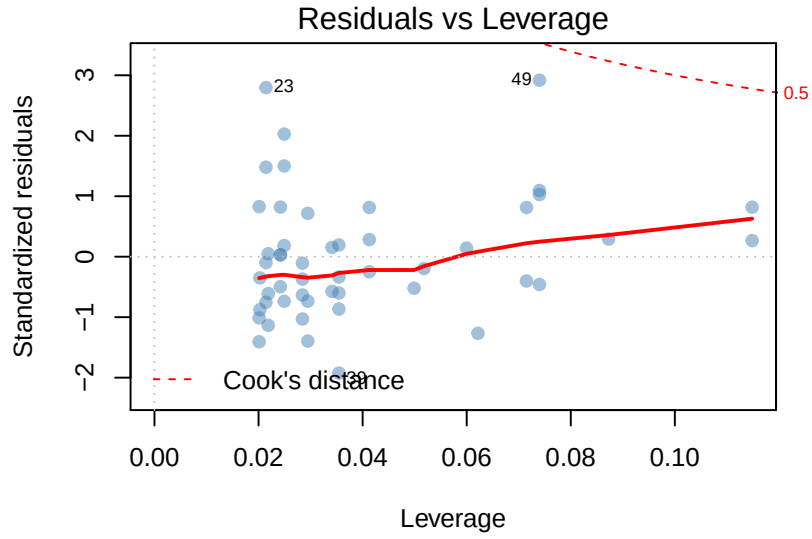
```
# Layout parameters (assignment saves previous values)
savePar <- par(mfrow=c(2, 2))
fit <- lm(dist ~ speed, data = cars)
plot(fit, pch = 19, col = adjustcolor("steelblue", 0.5), lwd = 2)
par(savePar)
```

Specialized `plot()` functionality: plotting a least-squares fit





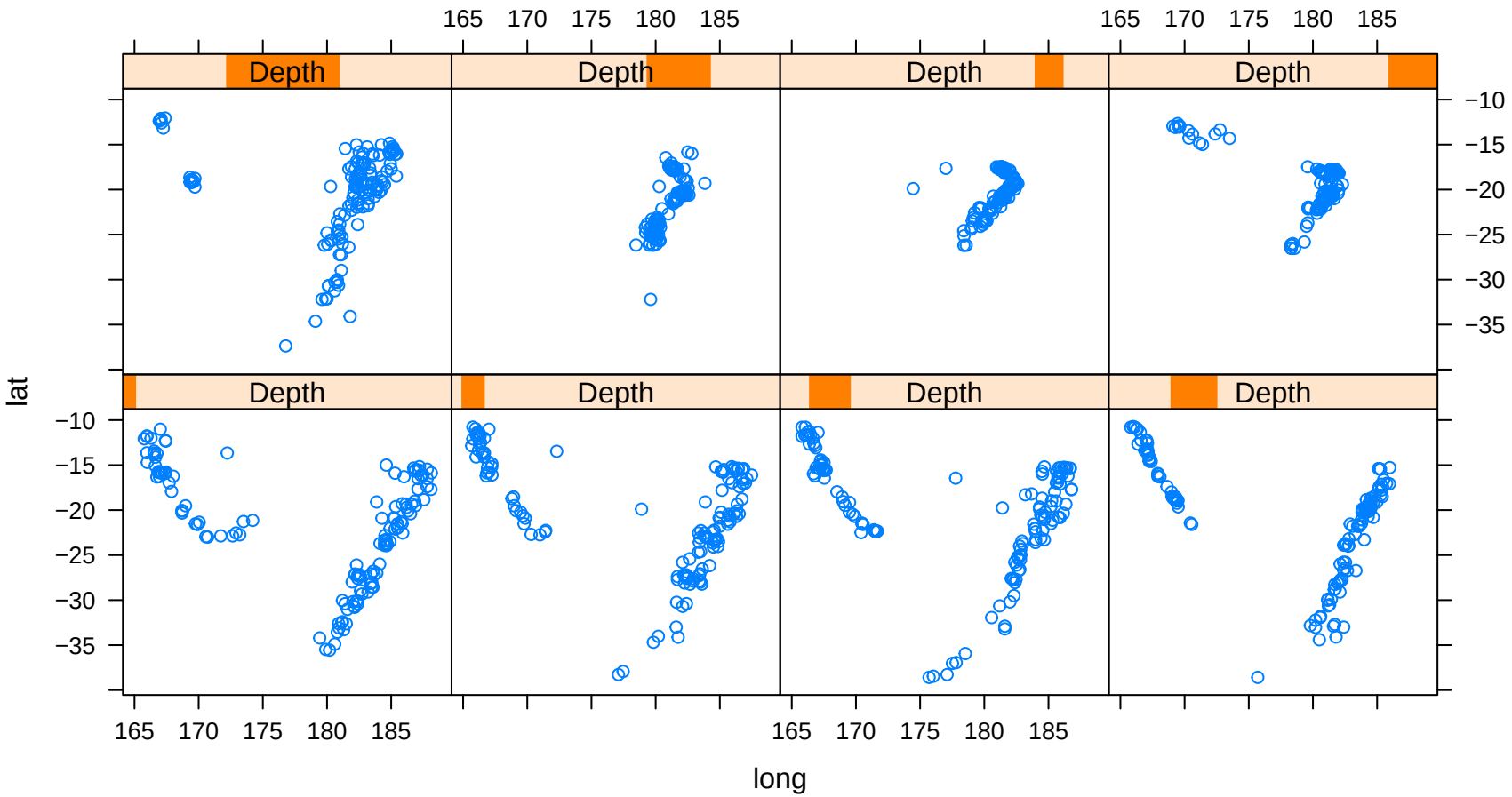




Powerful plotting packages built on top of it. E.g. `lattice`

```
library(lattice)
# Tonga Trench Earthquakes
Depth <- equal.count(quakes$depth, number=8, overlap=.1)
xyplot(lat ~ long | Depth, data = quakes)
```

Powerful plotting packages built on top of it. E.g. `lattice`



Advantages:

- it really is a very simple model
  - simple layout, simple graphics, little complexity
- simply add to the plot displayed
- very flexible, can easily create new displays
- some very powerful plotting functions (e.g. `coplot()`). Other plot functions (e.g. `pairs()`) also accept "panel functions"
- embedded in `S` (`R`) for decades, lots and lots of packages and new graphical displays are built on top of base `graphics`
- rich graphical systems have been built on top (e.g. `lattice` with its `xyplot()`, `dotplot()`, `barchart()`, `stripplot()`, etc.)
- many functions are generic, e.g. `plot()` and hence can be specialized to different data structures. This simplifies plotting for the user/analyst